SOT: Compact Representation for Triangle and Tetrahedral Meshes

Topraj Gurung and Jarek Rossignac
School of Interactive Computing, College of Computing,
Georgia Institute of Technology, Atlanta, GA

ABSTRACT

The Corner Table (CT) represents a triangle mesh by storing 6 integer references per triangle (3 vertex references in the Vertex table and 3 references to opposite corners in the Opposite table, which accelerate access to adjacent triangles). The Compact Half Face (CHF) representation extends CT to tetrahedral meshes, storing 8 references per tetrahedron (4 in the Vertex table and 4 in the Opposite table). We use the term *Vertex Opposite Table* (*VOT*) to refer to both CT and CHF and propose a sorted variation, *SVOT*, which is inspired by tetrahedral mesh encoding techniques and which works for both triangle and tetrahedral meshes. The *SVOT* does not require additional storage and yet provides, for each vertex, a reference to an incident corner from which the star (incident cells) of the vertex may be traversed at a constant cost per visited element. We use the corner operators for querying and traversing the triangle meshes while for tetrahedral meshes, we propose a set of powerful wedge-based operators. Improving on the *SVOT*, we propose our *Sorted Opposite Table* (*SOT*) variation, which eliminates the Vertex table completely and hence reduces storage requirements by 50% to only 3 references per triangle for triangle meshes and 4 references and 9 bits per tetrahedron for tetrahedral meshes, while preserving the vertex-to-incident-corner references and supporting the corner operators and our wedge operators with a constant average cost. The *SVOT* and *SOT* representation work on manifold meshes with boundaries.

Keywords: *Modeling, Triangle Meshes, Tetrahedral Meshes, Data Structures, Storage, Meshing, Polyhedra, Geometry Compression*

1. INTRODUCTION

1.1 Problem

Unstructured triangle and tetrahedral meshes are used in numerous applications, including finite element analysis [1-3], interpolation of samples [4], shape reconstruction [5], and medical image analysis [6, 7].

A variety of data structures and operators have been proposed [8-13] for storing the connectivity of triangle and tetrahedral meshes and for caching additional information that accelerates common queries and supports efficient traversal operators. In some applications, typical meshes contain millions of triangles or tetrahedra [14, 15] and this complexity continues to increase. In such applications, when the amount of fast-access memory available is limited, it is important to **reduce the storage cost** associated with these data structures.

Several triangle and tetrahedral mesh compression schemes have been proposed [16, 17]. Some support progressive refinements [18] or streaming [19-21]. Unfortunately, the compressed format they offer is not suitable for traversing, analyzing, simplifying [10, 11, 22, 23], refining [24], or improving [25-27] the mesh. Thus, an effective representation scheme is needed that provides efficient support for **random access operators that traverse the mesh** by accessing neighboring elements and that may be constructed efficiently from other (possibly compressed) formats and updated quickly to reflect mesh modifications.

1.2 Background

To place the proposed representations in a broader context, we briefly discuss a spectrum of representation schemes.

Shapes may be specified **procedurally** by a sequence of shape creation, editing, or merging operations [28-31]. Storing such a recipe as an implicit function [32, 33], as a collection of features (swept regions, pockets, protrusions), as an unevaluated Constructive Solid Geometry [34] model, or as its non-regularized generalization [35] makes it difficult to analyze the resulting shape, even though fast techniques exist for rendering it directly from some of these representations [36, 37].
Hence, modeling systems usually offer algorithms that **evaluate** the recipe by converting it into an **explicit** representation, which may be discrete or continuous. **Discrete** representations include point-clouds and ray-representations, which do not completely define the interior of the shape. **Continuous** representations decompose the shape into simple **cells**. Axis-aligned decompositions (such as voxels and K-D trees) provide poor approximations of the shape, unless a prohibitive number of cells are used.

A decomposition of a shape into a union of possibly curved, not necessarily simply-connected, and relatively open cells of various dimensions (points, edges, faces, volumes) may be performed through a recursive identification of geometric singularities (sharp edges and vertices) [Collins, 1976 and Whitney, 1957]. Furthermore, some of the cells may be explicitly identified as being part of the complement of the shape, hence providing a simple way of specifying shapes that are not topologically closed and that may contain cracks, such as a disk with a scratch (removed interior edge). Such models may be stored as a Selective Geometric Complex (SGC) [38] which is a collection of cells, each being a subsets of a possibly curved manifold and defined implicitly by that manifold, by its bounding cells, and by orientation flags. For example, a spherical cap may be defined by a sphere, by a bounding circle on that sphere, and by recording that the desired cap is to the left of the circle (where left is defined by orienting the normal of the circle in the surface). Hence, the representation of an SGC stores for each cell a reference to the supporting manifold and a list of references to its bounding cells, which we call **boundary** or **incidence references**. Many boundary representations (BReps) follow this principle, although for restricted topological and geometric

domains. We say that a cell b **bounds** a cell c if b is part of the boundary of c, relative to the closure of that manifold [38]. The **star** of a cell is the union of the point sets of the cells it bounds.

Restricting the supporting manifolds to linear ones yields the broad family of **non-manifold polyhedral** modeling schemes, where the supporting manifold of a cell does not need to be represented explicitly since it is implicitly defined as one that contains the bounding vertices. Restricting the cells to be the convex hulls of their vertices and assuming that all cells are contained in the shape yields a **simplicial complex**, which decomposes a shape into relatively-open cells: **0-cells** are the **vertices**, **1-cells** are the **edges** excluding their bounding vertices, **2-cells** are the triangular **faces** excluding their bounding edges and vertices, and **3-cells** are interiors of **tetrahedra** excluding their boundaries. We say that a cell is **incident** upon its bounding cells and that two k-cells are **adjacent** if they are incident upon the same (k-1)-cell.

One advantage of simplicial complexes over more general complexes [38] is the regularity of its representation: an edge may be represented by 2 vertex references, a face by 3, and a tetrahedron by 4. More generally, a k-cell is defined by k+1 vertex references.

The **k-graph** of a simplicial complex has nodes each representing a different k-cell and has a link between such two nodes when the corresponding k-cells are adjacent.

We say that the simplicial complex is a **k-mesh**, when it satisfied the following conditions:

(0) No self-intersection (proper imbedding): the intersection of any two different cells is empty (remember that we define cells as relatively open),

(1) No dangling cells (dimensional homogeneity): each m-cell with m<k is in the boundary of at least one k-cell,

(2) Manifold: the star of every m-cell with m<k is connected.

(3) Connected: the union of the cells is connected.

(4) Orientability: the orientation of all k-cells is compatible across shared boundaries.

The **valence** of a cell in a k-mesh is the number of k-cells incident upon it.

In this paper, we focus on representations of **2-meshes** (triangle meshes) and **3-meshes** (tetrahedron meshes). Models involving several connected components or that combine 3-meshes and 2-meshes may be represented as separate k-meshes, but the proposed representation does not support an explicit representation of non-manifold contacts between these k-meshes.

A k-cell of a k-mesh is **interior** if it has n adjacent k-cells. When a k-cell of a k-mesh is the only k-cell incident on a bounding (k-1)-cell B, we say that B is a **border**. An k-mesh with no interior k-cell is **narrow**.

We assume that the vertices are numbered from 0 to n_V–1 and that the vertex locations are stored in a **geometry table** G, where G[v] is the point where vertex number v is located. Other vertex attributes (density, color, normal) may also be stored, but are not discussed here.

The incidence information is typically stored as an (n+1)-tuple of cell-to-bounding-vertex references (integer indices) that identify the bounding vertices of an k-cell. Selecting an order for listing these references defines one of two possible **orientations** of the k-cell. For 3-meshes, we pick an orientation of each tetrahedron so that the vertices A, B, C, and D of the tetrahedron are listed in an order for which (AB×AC)•AD>0. For a 2-mesh, no such global orientation is defined, but when the triangle mesh represents the boundary of a solid s, the order (A, B, C) in which the vertex references are listed is chosen so that the vector AB×AC, when placed at (A+B+C)/3 points outwards of S. More generally, a triangle mesh is oriented when two triangles T1 and T2 are incident upon vertices A and B, then one of them must have one of the following 3 sequences of references (A,B,C), (C,A,B), or (B,C,A) and the other (B,A,D), (D,B,A), or (A,D,B). We assume that the mesh is orientable (for example that it is not a Moebius strip) and that the order of vertex references reflects this orientation.

We use the term **corner** when referring to this association of the integer reference of a vertex to one of its incident k-cells. We store these references in a single **V table**. The *k+1* references for each k-cell of a k-mesh are stored in consecutive entries in an order that respects the orientation of the k-cell. Hence, the V table has *(k+1)*t* entries, where *t* is the number of triangles in a *2-mesh* or the number of tetrahedra in a 3-mesh. Given an integer corner ID c, the function v(c) returns the vertex ID V[c] associated with that corner and the function t(c) returns the integer ID of the triangle for 2-meshes (or tetrahedron for 3-meshes) that contains the corner entry.

However, many applications that process triangle or tetrahedral meshes perform local connectivity queries. We distinguish three types of local queries:

- **Adjacency**: retrieve a particular k-cell adjacent to a given k-cell
- **Star**: retrieve a k-cell incident upon a given vertex
- **Order**: retrieve the next (k-1)-cell around a (k-2)-cell in a k-cell [39]

The G and V tables suffice for representing the k-mesh and to answer these queries. Unfortunately, the cost of retrieving such answers from the V table alone is linear ($O(n_T)$) in the number n_T of k-cells. So, most popular data structures for triangle meshes and for tetrahedral meshes cache additional adjacency, star, and order information to support these queries at constant cost.

When comparing the merit of different data structures, one considers **storage** (how many additional references per k-cell are cached), **performance** (how fast are these queries), and **simplicity** (how easy is it to use these queries when writing higher-level algorithms that traverse the mesh to compute various local or global properties).

For example, Brisson [40] stores **k-tuples** and for each k-tuple, stores a reference to a vertex and (k+1) references (called **swaps**) to other k-tuples. A k-tuple is associated with each combination of a k-cell A with one of its bounding (k-1)-cell B, with one of its bounding (k-2)-cell C, and so on. Hence, in a 3-mesh, there are 4×3×2 = 24 3-tuples and each one is associated with 4 references (one to a vertex and 3 to other k-tuples). This data structure requires 24×4=96 references per tetrahedron (rpt). Brisson's work is further discussed in Section 3.1.

A more compact representation of a k-mesh may store for each m-cell the m+1 references to its bounding (m–1)-cells which in turn store references to their bounding (m-2) cells and so on recursively and may also cache for each k-cell the k+1 references to its adjacent k-cells (set to nil when the adjacent cells do not exist) and also star references from each vertex to all incident k-cells. Symbolic depiction, of such representation for tetrahedral and triangle meshes are shown in Figure 1, where the number of references per cell is indicated next to the corresponding arrow. For example, in a tetrahedral mesh, a tetrahedron has 4 references to its faces and 4 references to its adjacent tetrahedra. A vertex has an average of 26 references to incident tetrahedra. A face had 3 references to its edges. An edge has 2 references to its vertices. Such representations would require around 20 references per tetrahedron (since the number of faces n_F is about twice the number of tetrahedra n_T, the number of vertices n_V is about one sixth of n_T and the number of edges n_E is about the same as n_T. Therefore we require $4n_T + 4n_T + 3n_F + 2n_E + 26n_V = 20n_T$). For the triangle representation scheme in Figure 1 (right) requires about 12 references per triangle (since the number of vertices n_V is about half of number of triangles n_T, number of edges n_E is about $3n_T/2$. Therefore we require $3n_T + 3n_T + 2n_E + 6n_V = 12n_T$).

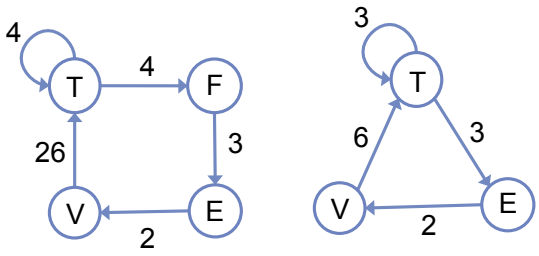

Figure 1: *Number of references required per tetrahedron (T), face (F), edge (E), vertex (V) (left). Likewise for triangle (T), edge (E), vertex (V) (right). The tetrahedral scheme requires about 20 references per tetrahedron, and the triangle scheme requires about 12 references per triangle.*

In contrast, we discuss here representations that provide the additional functionality, yet require only 3 references per triangle for 2-meshes or 4 references (+9 *service bits*) per tetrahedron for 3-meshes.

1.3 Contributions

The representations of k-meshes discussed here are based on the concept of a **corner**, which is the reference of a k-cell to one of its bounding vertices. Hence, a k-mesh with *t* k-cells has *t*(k+1) corners. Each corner of a triangle (resp. tetrahedral) mesh corresponds to two (resp. 6) n-tuples in Brisson's representation.

The Corner Table (CT) promoted by Rossignac et al. [41, 42] provides a simple and efficient representation of triangle meshes, storing 6 integer references per triangle (3 vertex references in the Vertex table and 3 references to opposite corners in the Opposite table). The Compact Half Face (CHF) representation proposed by Lage et al. extends CT to tetrahedral meshes, storing 8 references per tetrahedron (4 in the Vertex table and 4 in the Opposite table). We use the term *Vertex Opposite Table* (*VOT*) to refer to both CT and CHF.

For each ***corner*** c of each triangle (resp. tetrahedron), the ***Vertex Opposite Table*** (*VOT*) stores the references V[c] to the corresponding vertex and the reference O[c] to the opposite corners in an adjacent triangle (resp. tetrahedron) as shown in Figure 2, if one exists. It does not store any references from vertices to corners or to incident triangles (resp. tetrahedra).

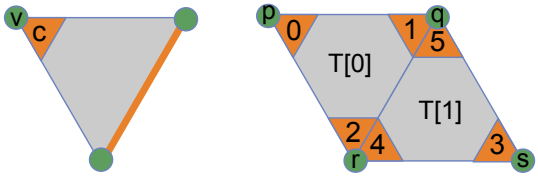

Figure 2: Left: *The orange edge e(c) is the opposite edge of corner c.* ***Right:*** *Corners 0 and 3 are opposite corners: O[0] = 3 and O[3] = 0, i.e. opposite edges e(0) and e(3) are the same.*

G	x	y	z
p	x_p	y_p	z_p
q	x_q	y_q	z_q
r	x_r	y_r	z_r
s	x_s	y_s	z_s

C	V[c]	O[c]	
0	p	3	T[0]
1	q	1	
2	r	2	
3	s	0	T[1]
4	r	4	
5	q	5	

Table 1: *VOT for* ***Figure 2, right****. Geometry table (left), Vertex Opposite table (right). Since corners 0 and 3 are opposite of each other, O[0]=3 and O[3] = 0. A corner c with no opposite corner (red) may be easily identified because we set O[c] = c.*

V**ertex-to-incident-corner references** are important in some applications because they provide constant cost access to the consecutive elements of the star of a vertex. To support them, we introduce a ***Sorted Vertex Opposite Table*** (***SVOT***) representation, which associates with each vertex v a reference to one of its incident corners V(c). Remarkably, SVOT caches this **vertex-to-incident-corner reference without additional storage**. This "trick" is accomplished by rearranging the order in which the n_T triangles (resp. tetrahedra) and their corners are stored in the VOT: When the mesh has n_V vertices, for any index $v<n_V$, integer v identifies the location of one of its incident corners, namely the first corner of the v^{th} triangle (resp. tetrahedron).

We provide **linear cost algorithms** for computing the O table of the *VOT* from the V table and an algorithm for converting a *VOT* into a *SVOT* that has linear time complexity.

Finally, we propose another extension of the *VOT,* which we call the ***Sorted Opposite Table*** (***SOT***). It further reduces the storage requirements to only **3 references per triangle** (resp. **4 references** and **9 *service bits* per tetrahedron)**, while preserving the direct vertex-to-incident-corner access. For triangle meshes, we store the SOT using $3n_T$ integers, whereas for tetrahedral meshes, we hide the *service bits* in the integer representation of the references and hence store the SOT using $4n_T$ integers (9 bits per 4 integers are used to store service bits). We eliminate the need for storing the Vertex table entirely. The vertex references v(c) of a corner is inferred from information stored in the Opposite table and the service bits.

To support the constructions of our SVOT and SOT and their use to support the traversal of the triangle mesh and the tetrahedral mesh, we have developed a set of powerful corner operators for triangle and tetrahedral meshes and **wedges operators** for tetrahedral meshes that extend the corner operators proposed by Rossignac et. al [43] and *half-edge* operators proposed by Lage et al. [44].

A wedge is the association of an edge with an incident tetrahedron and with a bounding vertex. As they operate on wedges, a set of our operators **mimic** the effect of corresponding triangle-mesh corner operators (next, previous, opposite, left, right, and swing) that operate on the triangle-mesh boundary of the star of the starting vertex of a wedge. We include the details of an **efficient implementation** of these operators, which have constant cost for VOT and SVOT, and average constant cost for SOT (where their expected cost is proportional to the valence of the base vertex).

Finally, we provide **examples** that demonstrate the ease of use of these data structures and operators for retrieving—at a constant (or average constant for SOT) cost per element—the **tetrahedra around an edge**, the **star of a vertex**, and the **connected component of the boundary** of the mesh.

An initial version of this work which discusses tetrahedral meshes only was presented at the SIAM/ACM conference on Solid & Physical Modeling (SPM) [45]. This extended version includes a thorough treatment of triangle meshes and several implementation details.

1.4 Structure of the paper

For clarity and simplicity, we first discuss our data structures and algorithms for triangle meshes in Section 2. Then, in Section 3, we explain how to extend them to tetrahedral meshes.

2. TRIANGLE MESHES

In this section, we discuss our novel representation for triangle meshes. After a brief review of prior art, we review the corner operators and explain efficient algorithms for constructing the O-table, then we introduce the SVOT, explain its construction, and show an example of its usage. Finally, we introduce the SOT representation and discuss its construction and usage.

2.1 Prior Art

Several data structures for polygonal meshes operate on edge-uses, which are each the association of an edge with a bounding vertex and with an incident face. Examples include Baumgart's *Winged-Edge* [46, 47]; Guibas and Stolfi's *Quad-Edge* [12], Mantyla's *Half-Edge* [48], and Lienhardt's *dart* [49]. Extensions of these schemes to non-manifold and non-regularized complexes include Weiler's *Radial-Edge* [50]. For additional discussion on data structures for simplicial complexes, we refer the interested reader to [51] and [52].

Although these techniques are suitable for representing triangle meshes and support efficient query and traversal operators [19], they require significantly more storage [39, 53] than the representations presented here, which have been optimized for triangle meshes. For example, the *quad-edge* stores 3 references per edge-use (1 to a vertex and 2 to other edge-uses), which amounts to $9n_T$ references.

More compact representations customized for triangle meshes include Campagna's et al. *Directed Edges* [54] and Kallmann and Thalmann's *Star-Vertices* representation [53], which, for each vertex, stores only the sorted list of references to its neighbors. For triangle meshes, *Star-Vertices* stores $4n_T$ references.

Other data structures have been introduced to reduce vertex cache misses [55] or to enable efficient traversal of primitives such as edges or triangles [56].

2.2 Corner Table/ Vertex Opposite Table

The *Corner Table* [42, 43, 57], which is the basis of the proposed solution here, represents the connectivity of a manifold triangle mesh of n_T triangles by two tables of $3n_T$ integers each. The V Table lists the triangle/vertex incidence, such that the 3 vertices bounding a triangle t are consecutive (V[3t], V[3t+1], V[3t+2]) and listed in an order that is compatible with a consistent orientation of the mesh. Hence, each

entry to the V[c] table represents a ***corner*** c associating a face (i.e. triangle) f with a bounding vertex. The O Table stores the integer reference of the ***opposite corner***, where an opposite corner is a corner in an adjacent triangle that shares the same opposite edge. The call the combination of the V Table and O Table the VOT. These ideas have been illustrated in Figure 2 and Table 1.

A set of ***corner operators*** provided to help in manipulation and traversal of the mesh (listed below and illustrated in Figure 3), may be trivially implemented from the information contained in these two tables:

```
int t(int c) {return int(c/3);}                          // triangle of c
int v(int c) {return V[c];}                              // vertex of c
int o(int c) {return O[c];}      // opposite or self if boundary corner
boolean b(int c) {return O[c]==c;}                       // border corner
int n(int c) {if ((c%3)==2) return c–2; else return c+1;}// next in t(c)
int p(int c) {return n(n(c));}                           // previous corner
int l(int c) {return o(n(c));}                           // tip on left
int r(int c) {return o(p(c));}                           // tip on right
int sr(int c) {return p(r(c));}                 // next (right) around v(c)
int sl(int c) {return n(l(c));}                 // next (left) around v(c)
```

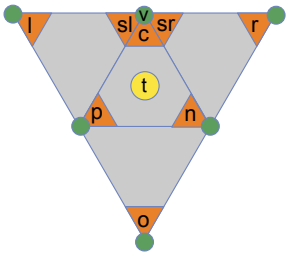

Figure 3: *The corner operators for a triangle mesh. For example, the integer ID of the orange corner n is obtained from the integer ID of corner c by the function call n(c).*

2.2.1 O Table Construction

The O Table does not need to be archived since it may be recomputed in linear time from the V Table when the V Table is loaded. To do so, for each corner c, we make an entry (v_1, v_2, c), where v_1=min(v(n(c)),v(p(c)) and v_1=max(v(n(c)),v(p(c)). We sort them lexicographically by (v_1, v_2). Pairs of consecutive entries, (v_1, v_2, c) and (v_1, v_2, d) identify opposite corners: O[c]=d and O[d]=c.

Sorting may be performed in expected linear time using hashing on (v_1, v_2). as demonstrated by Lage et al. [44] or in linear space and time using buckets [58], as proposed by Ueng and Sikorski [21, 59], where the entries are first sorted by v_1 into n buckets and then, one bucket at a time, using a temporary table of *n* sub-buckets, sorted by v_2. A similar approach was used by Rossignac and Borrel [RoBo93] to identify dangling edges in simplified meshes.

2.3 Improvement over VOT

In the following subsections, we discuss three improvements to VOT.

(i) Vertex to corner access: To provide a constant cost access, we reorder the triangles so that for the first n_V triangles, the first corner of the i^{th} triangle corresponds to the i^{th} vertex (see Figure 8). We call this data structure the Sorted Vertex Opposite Table (*SVOT*). We discuss this in sub-section 2.4.

(ii) 3 references per triangle: To further reduce storage, we discard the V Table. The resulting Sorted Opposite Table (*SOT*) stores only 3 references per triangle. We discuss this in sub-section 2.5.

(iii) Constant time operators: We use the same operators as the original corner table, except for the v operator (vertex operator). We discuss this in section 2.5.4.

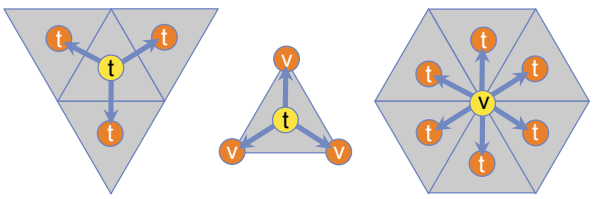

Figure 4: *Our data structure provides direct access to: i) adjacent triangle neighbors (left), ii) vertex references for a triangle (center), and iii) triangles incident on a vertex (right), in average constant time (per retrieved element)*

2.4 Sorted Vertex Opposite Table (SVOT)

2.4.1 Motivation

Various operations such as computing normals or curvature may be easily expressed by looping through or manipulating triangles or corners. For example, one may find all triangles that lie inside a given ball by visiting all triangles and accessing their corners and testing the corresponding vertices for inclusion in the ball. Temporary flags could be used to avoid testing the same vertex more than once. Similarly, one may find the triangle with the sharpest corner by looping through all triangles and for each one, by looping through its three corners.

However, some developers prefer to have **direct access from a vertex to its star** (incident triangles), because some of their algorithms operate directly on vertices (not through corners) or because some of their auxiliary data structures refer directly to vertices. A natural solution [44] is to add a **vertex-to-corner lookup table** C, such that C[v] contains the index of corner c, such that V[c]=v. This approach requires storing additional n_V references. The *SVOT* solution described below provides the same information though a constant cost function call c(v) and avoids storing the C table.

2.4.2 Proposed SVOT solution

To provide constant cost access to a corner c(v) for each vertex v, and this without additional storage, we reorder the triangles and their corners in the VOT so the corner c(v) incident upon vertex v may be simply computed as 3v. This mapping works for most meshes. However, for meshes with *narrow* components, we may need a slightly more complex special mapping, as shown in Figure 5 which we discuss in Section 2.4.4. An edge-connected component of a mesh is ***narrow*** if each triangle in the component has at least one *border vertex.* A *border vertex* is a vertex that lies on the boundary of a mesh.

2.4.3 SVOT construction algorithm

We explain here how to compute SVOT from the VOT in linear time. The process involves 2 steps:

1) Vertex to Triangle Mapping: Establish a **mapping,** M(t)=v, between each triangle t and a bounding vertex v so that **no two triangles map to the same vertex**.
2) V Table Sorting: Reorder the *VOT* based on the mapping.

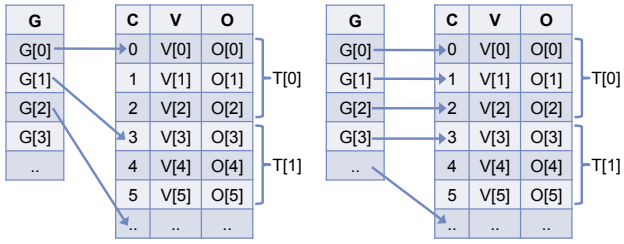

Figure 5: Left: *General SVOT mapping.* ***Right:*** *Special mapping. After the sorting, the V table may be discarded to obtain the SOT.*

2.4.3.1 Vertex to Triangle Mapping

The *mapping phase* involves three steps: (i) ***Initialization***, (ii) ***Traversal***, and (iii) ***Termination***.

To represent the mapping M() computed in the *mapping phase*, we use a temporary table M which stores the vertex number M[t] associated with triangle t. We use three arrays of auxiliary tables: **visitedV[v]** keeps track of visited vertices v, **visitedT[t]** keeps track of visited triangles t, **whichCorner[t]** stores the corner c in triangle t such that M[t(c)]=v(c).

We process an edge-connected component of the mesh. We traverse the Triangle Spanning Tree (*TST*). The TST is the set of triangles visited in depth first order from an initial triangle. As we enter a new triangle t through an edge e, we associate t with the reference to its ***tip vertex*** (the vertex in triangle t not bounding edge e), unless that vertex has already been associated with another triangle. This idea is similar to the association of the tip vertex of each *type-C* triangle in the Edgebreaker compression scheme [42] for triangle meshes. Unfortunately, unless we take special precautions, this simple idea may not always work, because the traversal may associate each triangle incident upon a vertex v with a vertex other than v, leaving some vertices **unmapped** to any triangle. To eliminate the possibility of unmapped vertices, we perform a special **initialization step**, which **guarantees** that this approach produces a *correct mapping*. A *correct mapping* is one where all vertices are mapped to triangles such that each vertex is associated with a unique triangle.

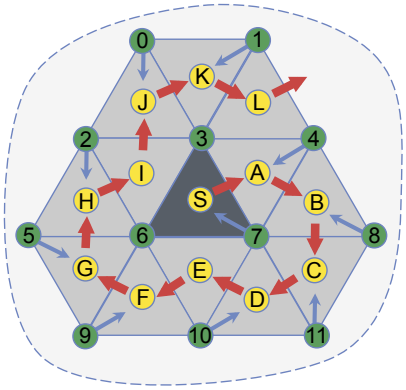

Figure 6: *Depth first traversal of triangle mesh starting from seed triangle S. Red arrows represent traversal order, blue arrows represent mapping of vertices to triangles. Notice vertices 6 and 3 are marked as visited in the initialization step. Also because a non-border seed triangle S is chosen (i.e. all its vertices are interior), triangles E and I are unmapped.*

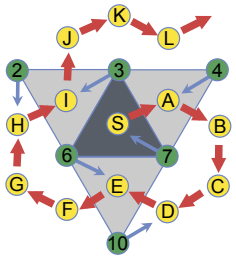

Figure 7: *A portion of* ***Figure 6****. In* ***Figure 6****, vertices 3 and 6 are unmapped, and triangles E and I are unmapped too. We introduce mappings (6,E) and (3,I). Due to the nature of depth first traversal, and a internal triangle S, this mapping is always possible.*

(i) ***Initialization:*** During the initialization step, we pick a ***seed*** triangle S so that **none of its vertices bound a border edge**.

Let c be the first corner of the seed triangle S. We set all entries in M[] to be -1 denoting unmatched triangles. We set M[S]=v(c) and **mark (as visited) all vertices of S**.

In Figure 6, the seed triangle S is the dark gray triangle with label S. S is bounded by vertices 7, 6 and 3 and Vertex 7 is v(c). Vertices 7, 6, 3 are marked as visited and M[S] = 7.

Note that this approach assumes that a suitable seed exists. Finding S, when it exists is trivial. The approach proposed above works for edge-connected components of meshes that are not *narrow*. It picks as seed a triangle with no border vertex. For *narrow* components of meshes, we use a slightly modified solution in section 2.4.4. For multiple edge-connected components, we need multiple seed triangles. After selecting a seed triangle S, we mark S and start a depth first traversal of the triangle spanning tree with S as root and t(l(c)) as the first child, where c is the first corner of S.

In the initialization step, in Figure 6, visitedV[7], visitedV[6], visitedV[3] and visitedT[S] are set to as TRUE. E.g., if the corner of S bounded by vertex 7 was 30 then whichCorner[S] would be assigned 30.

(ii) ***Traversal:*** During the traversal step, in our depth first order traversal, we reach a new unvisited triangle t by arriving from a parent triangle through the opposite edge e(b) of a corner b (of triangle t). We mark t as visited. If the vertex v(b) has not yet been visited, we mark it as visited, set M[t]=v(b), and store in whichCorner[t(b)] corner b.

For example, in Figure 6, the first triangle visited after triangle S is triangle A. We arrive at triangle A through edge (3,7). Let b be the corner bounded by vertex 4. Since vertex 4 was previous not visited, therefore, M[A] = 4 and whichCorner[A] = b.

(iii) ***Termination*****:** We match the two triangles *E* and *I* Figure 7 incident upon seed S (which are t(l(c)) such that c is not the first corner of S) with 2 vertices 6 and 3 of S (which are v(c) such that c is not the first corner of S), as shown in Figure 7. Given the precaution we took with the selection of the seed triangle as an internal triangle, then S has three adjacent triangles, the other 2 triangles *E* and *I* adjacent to S have been reached while coming from triangles other than S and hence will not be associated with a vertex (since their tip vertex 6 and 3 was mapped to S during initialization and is no longer available to be associated with them).

Since we perform a depth-first traversal starting from S, we visit all edge-connected triangles, which implies that we visit all the vertices. Other than the *seed triangle* S and its incident vertices, which we addressed above, each time we visit a vertex, we map it to the visiting triangle. Therefore each vertex is mapped to a unique triangle.

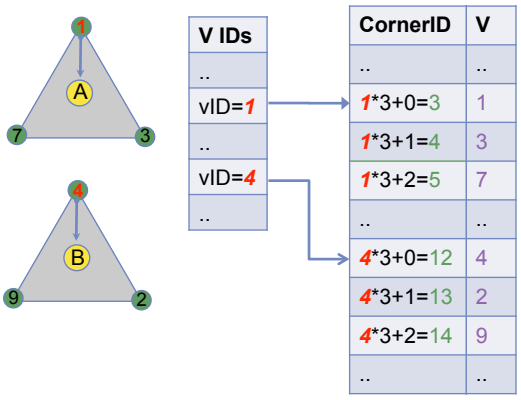

Figure 8: *Vertices 1 and 4 are mapped to triangles A and B. Triangle A consists of vertices (1,3,7) and triangle B of (4,2,9). After sorting in SVOT, triangle A is defined by corners (3,4,5) and triangle B by corners (12,13,14) as vertex 1 maps to triangle at 1st location and vertex 4 maps to triangle at 4th location.*

2.4.3.2 Sorting the V Table

We write the sorted triangle into a new copy of the V table ensuring that triangle M[t]=v is listed as triangle number v and performing a cyclic permutation of the corners of each triangle so that the remembered corner stored in whichCorner[t] is listed as the first corner of t. Since we have changed the ordering of the entries of the V table, the O table references need to be updated, so we re-compute the O Table as explained in section 2.2.1. Figure 8 illustrates the resulting SVOT.

The sorting discussed above requires $O(n_T)$ time and $O(n_T+n_V)$ temporary space for marking triangles and vertices. The sorting needs to be performed only once, since its result (i.e. the sorted V-table) may be archived for future uses. This sorting requires $O(n_T)$ time instead of the traditional $O(n_T \log n_T)$ that is associated with sorting, as this sorting is a permutation where each element knows the location of the bin it wants to be in the sorted order.

2.4.4 Special cases of narrow components

For each edge-connected *narrow* component, we choose any triangle as the seed S. We place S as the first triangle in the V table. Let the three vertex references for S be v_0, v_1 and v_2. We swap the vertices in the geometry table. We swap vertex v_0 with the 0^{th} entry in the geometry table, vertex v_1 with the 1^{st} entry in the geometry table and likewise for v_2. We then reorder the rest of the triangles as described in the construction section for the general case. Now, the vertex-to-corner mapping is as follows: for $i<3$, the i^{th} vertex maps to the i^{th} corner. For $i\geq 3$, the i^{th} vertex maps to the $((i-2)*3)^{th}$ corner. Correspondingly, for meshes with *m narrow* edge-connected components, we can place each *seed* triangle for each component as the first *m* triangles in the V Table. This mapping has been shown in Figure 5, right.

2.4.5 Traversing the star

Given an integer reference v to a vertex, the *SVOT* gives us direct access to the corresponding corner c(v), using:

```
int c(int v) {return v*3;}                    //for triangle meshes
```

The i^{th} vertex is mapped to the $3*i^{th}$ corner in the SVOT. Notice, in Figure 8, in the SVOT, vertex 1 is located in the corner location 3*1=3, and vertex 4 is located in the corner location 3*4=12. To visit the star of the i^{th} vertex, we simply call star(3*i) and we can traverse all incident triangles on a vertex by iteratively using the swing right and swing left corner operators. The star(c) function, which assumes that vertex v(c) is interior, is listed below.

```
void star(int c) {                                  //star traversal
 int sc = c;                                       //starting corner
 do {
  c = sr(c);                                 //visit swing right corner
  process(t(c));                               //process the triangle
 } while(c!=sc); }                           //back to starting corner
```

When v(c) maybe a border vertex, we need to keep track of boundary corners and use the swing left corner operator too.

```
void star(int c) {                                  //star traversal
 int sc = c;                                       //starting corner
 process(t(sc));                        //process the starting triangle
```

```
do {
  c = sr(c);                          //visit swing right corner
  if(c==sc || c==-1) break;          //if boundary or starting corner
  process(t(c));                      //process the triangle
} while(true);                        //loop
if(c==-1) {                           //if boundary corner
  c = sc;                             //visit swing right corner
  do {
    c = sl(c);                        //visit swing left corner
    if(c==sc || c==-1) break;        //if boundary or starting corner
    process(t(c));                    //process the triangle
  } while(true); }}                   //loop
```

The corner operators for the Corner Table work without modification on the *SVOT*.

2.5 Sorted Opposite Table (SOT)

The *VOT* and our *SVOT* variation each store 6 references per triangle (3 to vertices, 3 to opposite corners). We discuss here an approach to reduce this storage to 3 references per triangle.

2.5.1 Construction of SOT:

To construct the SOT, we first construct the *SVOT*. We then obtain the *SOT* by eliminating the V Table in the *SVOT*.

Note that the *SOT* maintains the property of the *SVOT* where the i^{th} vertex v_i maps to the i^{th} triangle T_i and the corner $c_i=3*i$ is incident on the vertex v_i.

2.5.2 High level description

The *Sorted Opposite Table* (*SOT*) uses the same O Table as the one produced in *SVOT,* **but does not store the V table at all.** Consequently, the resulting *SOT* contains no references to vertices. How then is it possible to find vertex references v(c) of a corner c? The solution comes from a combination of three ideas.

1) Because the O Table is sorted in the *SVOT*, to each vertex v corresponds a ***matching*** triangle t_V=v of which the first corner is incident on v. Note that such triangles are easily **recognized** because their index t is less than the number n_V of vertices. A slightly modified mapping relation is used for *narrow* components (See Section 2.4.4 for *narrow* components).

2) By construction of the SVOT, in the **star** of every vertex v, there is a ***matching*** triangle t_V = v.

3) Starting from any corner c, we can use the star function provided above to visit the *star* of its vertex v(c), even though we do not yet know the index of v.

The idea is to traverse the ***star*** (see section 2.4.5) of v(b) to determine the corners b_i incident on v(b). We must do that of course without knowing v(b), since v(b) is the desired result. Traversing the star is possible by using the swing right and swing left corner operators as these corner operators require only the O Table (i.e. connectivity information, not the V Table) of the SOT. The traversal stops when we find a ***matching*** triangle T<n_V.

Finding the vertex reference can require that we visit at most d triangles, where d is the valence of the vertex v. Since the valence of a vertex on a triangle mesh is, on average, 6, therefore, we need to visit, on average, 3.5 triangles.

2.5.3 Corner operators on SOT

The implementation of all corner operators from the *VOT* remain the same in SOT, except for v().

The v() operator in SOT traverses the star of vertex v(c) without knowing v(c) to determine the vertex reference. The code for the v(c) function is listed below.

```
int v(int c) {                                  //vertex id
  int sc = c;                                   //save starting corner
  do {
    if(t(c)<nV && c%3==0) return t(c);          //first corner of triangle
    int cc = c;                                 //current corner
    c = sr(c);                                  //visit swing right corner
  } while(c!=sc || c!=cc);      //back to starting or boundary corner
  c = sc;                                       //restarts from starting corner
  do {
    if(c/3<nV && c%3==0) break;                 //first corner of triangle
    int cc = c;                                 //current corner
```

```
  c = sl(c);                          //visit swing left corner
 } while(c!=cc);                      //boundary corner
return t(c);}                       //error, no vertex id found
```

Starting from a corner c, v() pivots clockwise and then, if necessary, counterclockwise around v(c) using the sr(c) and sl(c) corner operators. It stops and returns t(c) if it finds a corner c such that $t(c) < n_V$.

The small overhead cost of the new v() function, detailed below is often justified by the reduction of storage, and hence of page faults.

Our vertex operator uses the O Table and the star traversal idea explained in section 2.4.5.

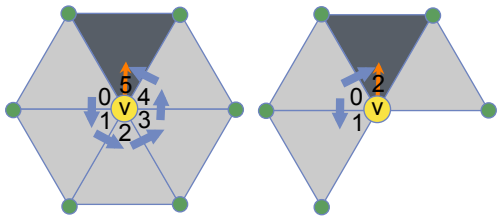

Figure 9: *To determine the vertex ID of the vertex v, we traverse the incident triangles by using the swing corner operator, sr or sl. In the SOT, one of the incident triangles has the i^{th} vertex mapped to the i^{th} triangle's first corner (yellow vertex and orange arrow).*

2.5.4 Special cases of narrow components

The special cases of narrow meshes generalizes from the *SVOT* (Section 2.4.4) to the *SOT*. For narrow meshes, the vertex references of a corner is found by traversing the star of a corner until a vertex-to-corner mapping is found. The mapping provides the vertex reference as defined in Section 2.4.4.

2.6 Summary

We have provided a linear cost construction algorithm that starts with the popular triangle-vertex-incidence information stored in the V table. And then computes the O tables, sorts the V and O tables to provide constant cost access to the star of each vertex and finally discards the V table to provide the the SOT representation, which stores only $3n_t$ references.

3. TETRAHEDRAL MESHES

We extend the SVOT and SOT for triangle meshes to tetrahedral meshes. Most of the ideas developed above for triangle meshes extend trivially to tetrahedral meshes. The VOT structure, which the SVOT and SOT rely on, requires 8 references (4 for vertex references, 4 for opposite references) for tetrahedral meshes as opposed to 6 references (3 for Vertex references, 3 for Opposite references) for triangle meshes. To accommodate the traversal of tetrahedral mesh elements, we introduce the wedge operators.

3.1 Prior Art

Designing a set of low level operators on the cells of a tetrahedral mesh is not trivial, because simple queries, such as "what is the next tetrahedron" or "what is the next face" need a context: "next around what?". That context is usually more complex than a single element. For example, it is not clear what face is the "next" face after face f in an incident tetrahedron. Nor is it clear what face is the "next" face after face f around a bounding edge. The notion of a vertex-use or edge-use (which suffice to disambiguate such queries for oriented triangle meshes) may help, but still lacks specificity and generality. For example, an edge is used by its two vertices, by its k incident faces, and by its m incident tetrahedra.

Brisson [40] has provided an elegant tool for distinguishing all such ***cell-uses*** and for specifying unambiguously the "next" operator through which the ordering information is obtained. Brisson's work was aimed at complexes in arbitrary dimension. The restriction of his approach to tetrahedral meshes is based on ***4-tuples***, which each list 4 cells, one of each dimension, so that each cell in a 4-tuple is incident upon all cells of inferior dimension. Hence, one such 4-tuple, say (v,e,f,t), is defined by the references to a particular vertex, v, to an edge e incident on v, to a face f incident on e, and to a tetrahedron t incident on all three. Given such a 4-tuple, Brisson defines four "next" operators, one per dimension. Each one is its own inverse. Hence he calls them ***swap*** operators. Let us examine them one by one.

0) $swap_0$(v,e,f,t) returns the 4-tuple (v',e,f,t), which is unique, since there are only two vertices, v and v', with the same three incident cells (e,f,t). Hence, $swap_0$ may be used to swap between the two end-vertices of an edge.

1) $swap_1$(v,e,f,t) returns the 4-tuple (v,e',f,t), which is unique, since there are only two edges, e and e', that share a bonding vertex v and have the same two incident cells (f,t). Hence, $swap_1$ may be used to swap between adjacent edges of a face. By cascading $swap_0$ and $swap_1$ calls, one may walk around the boundary of a face f. The choice of which incident tetrahedron t is used defines the direction of the circular walk and hence endows f with an orientation.

2) $swap_2$(v,e,f,t) returns the 4-tuple (v,e,f',t), which is unique, since there are only two faces, f and f', that share a bounding edge e and are

bounding the same tetrahedron t. Combinations of $swap_0$, $swap_1$ and $swap_2$ provide a tool for visiting all of the faces of a tetrahedron.

3) $swap_3$(v,e,f,t) returns the 4-tuple (v,e,f,t'), which is unique, since there are only two tetrahedra, t and t', that share a bounding face f. Combinations of $swap_2$ and $swap_3$ rotate around edge e. The direction of rotation is defined by the choice of vertex v.

Unfortunately, there are $24n_T$ such 4-tuples in a mesh of n_T tetrahedra and we would need to store for each one of them the vertex references and 4 other IDs (one per *swap* operator) to define the results of the *swaps*. Hence the total storage cost for the 4-tuples and their *swap* references would be $120n_T$ references (typically integers).

Note that not all cell-references are required to provide the context needed for a *swap*. For instance, $swap_0$ requires only V and E; $swap_1$ requires only V and F; $swap_2$ requires only E and T; and $swap_3$ requires only F. In general, $swap_d$ swaps cell of dimension d and only needs as context the cells of dimension d–1 and d+1, when they exist. Based on this observation, Rossignac [Ros94] proposed a more compact representation called *NAIL* (abbreviation for Next cell Around cell In cell List), which associates with each cell a table (one dimensional table for vertices and tetrahedral and two-dimensional table for edges and faces), which, uses this reduced context as index and provides the corresponding cell returned by Brisson's swap.

Several data structures customized for tetrahedral meshes [3, 8, 9, 12, 13, 60-64] provide a more compact representation. They, or their variations, have been used for mesh generation and processing [65-70].

Several techniques were proposed for compressing tetrahedral meshes [16, 17, 71], for streaming them [20, 27, 72], and for transferring refinements or simplifications [18, 23, 73]. But random access operators that traverse the compressed mesh (such as those developed for triangle meshes [74]) are not supported in these approaches.

Several representation schemes were developed to support multi-resolution tetrahedral meshes [22, 52, 75-81]. In more direct relevance to our work, Szymczak and Rossignac [82] propose the Grow&Fold compression algorithm for tetrahedral meshes. They compute a ***Tetrahedron Spanning Tree*** (***TST***) and encode it using 3 bits per tetrahedron. Except at the root, the traversal of the *TST* enters a tetrahedron T by a face f. Each one of the 3 bits associated with T corresponds to a different face of T (excluding f) and indicates whether T has a child in the *TST* incident upon that face. Because the *TST* does not encode the complete connectivity, they also store two bits per border face of the *TST* to control a folding process that reconstructs the full connectivity. These bits indicate whether the face is a border face of the mesh and, when not, select one of its edges for folding. Since there are roughly $2n_T$ such border faces, their scheme requires about $7n_T$ bits to encode the connectivity of the mesh. It is impractical to require a full traversal of the *TST* to identify the parent of a tetrahedron and to require the executing of the folding algorithm to recover the references of the other two adjacent tetrahedra. Hence, these references must be cached. Furthermore, a tetrahedron may have 0, 1, 2, or 3 children in the *TST*. We must be able to locate these children in constant time, without having to traverse the rest of the *TST*. Thus, we cannot use their compressed format.

Weiler et al. [83] encode tetrahedral meshes in strips. Using a greedy stripification algorithm, they obtain an average of 4.3 tetrahedra per strip. They also discuss a variation that builds longer strips by allowing duplicate vertex entries in a strip. A tetrahedral strip is stored as an ordered list of vertex references such that any 4-tuple of consecutive vertices in a strip bound a tetrahedron. Each tetrahedron in a strip is face-adjacent to its predecessor and successor in the strip (when they exist). A strip with k tetrahedra has 3+k entries in V (one per vertex) and has 2+2k external faces, for which they store opposites in the O-table. Hence, to produce a regular structure, they use 3 tables, each of size (3+k), one for the vertices, and two for the opposites, resulting in storage cost of 3(3+k) per strip, which results in $3n_T + 9n_S$ total storage, where n_T is the number of tetrahedra and n_S is the number of strips. Since there are 2+2k external faces, but 6+2k locations in the opposites table, 4 locations in the opposites table do not contain any information. A bit per corner is used to identify the beginning and ending of strips. Assuming 4.3 tetrahedra per strip, the total storage cost is $(3n_T+9(n_T/4.3))= 5.1n_T$, i.e. an average of 5.1 references per tetrahedron. Their variation that allows repetition of vertices requires 4.6 references per tetrahedron.

3.2 Corner Table Extension (VOT)

The Corner Table has been extended by Bischoff and Rossignac [27] and by Lage et al. *Compact Half Faces* (*CHF*) [44] to tetrahedral meshes. The VOT requires 8 references per tetrahedron (4 for vertex references and 4 for opposite corners). An index to these tables identifies a particular **corner** of a particular tetrahedron. Therefore, the V and O tables each have $4n_T$ entries.

As was done for the triangle meshes, the corners of each tetrahedron are ***consecutive*** in the *VOT* (the 4 corners for the i^{th} tetrahedron are stored at entries 4i+j, where j = 0,1,2,3) and are listed in an order that is consistent with the orientation of the tetrahedron (the vertices of corners j=1,2,3 appear counter-clockwise from the vertex of corner j=0). The order is important because the next wedge operator requires a consistent orientation. Figure 10 (left) illustrates the corners and the orientation of a tetrahedron. When two tetrahedra share a face, the two corners, b and c, that do not lie on the shared face are opposite and we cache this relation: O[b]=c and O[c]=b.

We illustrate the VOT on a mesh of two tetrahedra in Figure 10 (right), where we have numbered the corners. The corner pairs (1,5), (2,7) and (3,6) each share the same vertices b, c and d respectively. Corners 0 and 4 are opposites of each other as they share the same opposite face (b,c,d). The other corners do not have opposites. For each such border corner c, we set O[c]=c.

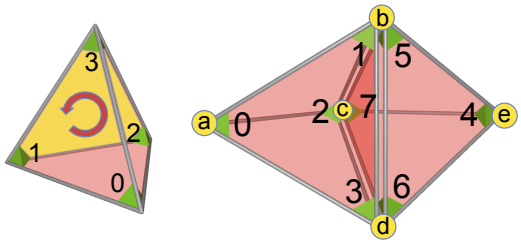

Figure 10: *The corners are shown as small green tetrahedra.* ***Left****: The yellow face f(0) is the opposite face of corner 0. Corners 1, 2 and 3 are assumed to be counterclockwise orientation relative to corner 0.* ***Right****: Corners 0 and 4 are opposites: O[0]=4 and O[4]=0. The two tetrahedra have been slightly shrunk for clarity, but are in fact adjacent to each other: f(0)=f(4).*

G	x	y	z
a	x_a	y_a	z_a
b	x_b	y_b	z_b
c	x_c	y_c	z_c
d	x_d	y_d	z_d
e	x_e	y_e	z_e

C	V[c]	O[c]	
0	a	4	T[0]
1	b	1	
2	c	2	
3	d	3	
4	e	0	T[1]
5	b	5	
6	d	6	
7	c	7	

Table 2: *VOT for* ***Figure 10, right****. Geometry Table (left), Vertex Opposite Table (right), O[0]=4 and O[4] = 0. Corners with no opposite corners (red) are identified by O[c] == c. For clarity, we use characters (a, b, c, ...) here to represent vertex IDs which are in fact positive integers (0, 1, 2, ...)*

3.2.1 O Table Construction

The construction of the O Table follows the one presented in Section 2.2.1 but here, we use four-tuples $\{v_1,v_2,v_3,c\}$ computed as follows, s=v(n(c)), t=v(n(n(c))), u = v(n(n(n(c)))) and $\{v_1,v_2,v_3\}$ = sorted {s,t,u}. Hashing [44] or bucket sort [59] maybe used to sort the tubles and provide opposite pairs. The bucket sort requires three levels here, but still has linear cost.

3.3 Corner Operators

We use upper case for tetrahedral mesh ***corner operators***.

```
int T(int c) {return d4(c);}                    // tetrahedron of c
int N(int c) {return fc(c)+m4(m4(c)+1);}        // next corner in T(c)
int P(int c) {return fc(c)+m4(m4(c)+3);}        // previous corner
int V(int c) {return V[c];}                     // vertex of c
int O(int c) {return O[c];}                     // opposite corner
boolean B(int c) {return O(c)==c;}              // f(c) is a border face
```

They are based on the following ***auxiliary bit-manipulation operators*** (similar to [44])

```
boolean even(int c) {return ((c&1)==0);}        // c is even
int m4(int c) {return c&0x3;}                   // c modulo 4
int d4(int c) {return c>>2;}                    // c divided by 4
int x4(int t) {return t<<2;}                    //t multiplied by 4
int fc(int c) {return x4(d4(c));}               // first corner of t(c)
```

3.4 Wedge Operators

To traverse the mesh and to access the various elements (vertices, edges, faces, and tetrahedra) and their neighbors in an orderly fashion, we use the concept of a ***wedge***, which is the association of a ***base vertex*** v with an incident edge e and an incident tetrahedron t. It corresponds to a the *half-edge* [44]. In our figures, a wedge w defined by the triplet (v,e,t) is shown as a colored arrow along pointing e away from v. For simplicity, w.a denotes the **starting corner** of w and w.b its **ending corne**r.

We define 10 wedge operators (Figure 11) and use lower case names for them. Consider an interior vertex v. The boundary of its star is a triangle mesh M homeomorphic to a sphere. To each wedge w=(v,e,t) corresponds a corner c of M. We have named some of our wedge

operators so that they preserve this correspondence. For example, as shown in Figure 11 (top right), n(w) (in green) corresponds to n(c), p(w) (in blue) corresponds to p(c). The triangle corners are not shown to avoid clutter, but are on the face to which the corresponding wedge arrow points. Similarly, as shown in Figure 11 (bottom right), o(w) (in dark gray) corresponds to o(c), l(w) (in cyan) corresponds to l(c) and r(w) (in magenta) corresponds to r(c). Note that in this text, we differentiate between triangle mesh corner operators and the corresponding tetrahedral mesh wedge operators by the context or the type of the parameter, i.e. c (corner) or w (wedge) (even though in our implementation, both are integers).

Several wedge operators do not have corresponding corner operators. The ***mirror wedge*** m(w) returns Brisson's $swap_0$. The ***cross wedge*** operator k(w) returns the wedge whose edge is not adjacent to the edge of w, and that appears to go left, when seen from an observer aligned with (the arrow used to show) w.

We **do not store wedges** explicitly, since storing them would require a significant amount of memory. Instead, we represent a current wedge w by an ordered pair (w.a, w.b) of two references to corners of the same tetrahedron. In Figure 11 (top left), the red wedge is defined by the green corners (a,b). Clearly, such an ordered pair of corners defines the triplet (v,e,t) of a staring vertex v, a supporting edge e, and an incident tetrahedron t.

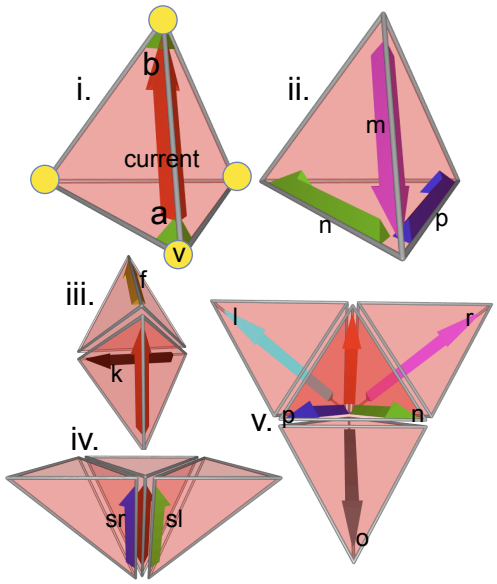

Figure 11: *Wedge w is shown as a red arrow.* ***(i)*** *red wedge (a,b) goes from corner a to corner b.* ***(ii)*** *next wedge n(w) in green, previous wedge p(w) in blue, mirror (reversed) wedge m(w) in magenta.* ***(iii)*** *cross wedge k(w) in black (back), forward wedge f(w) in yellow.* ***(iv)*** *swing right wedge sr(w) in blue, swing left wedge sl(w) in green.* ***(iv)*** *opposite wedge o(w) in dark gray, left wedge l(w) in cyan, right wedge r(w) in magenta, next wedge in green, previous edge in blue.*

Our *next*, *mirror* and *opposite* wedge operators have the same functionality as the *next*, *mate*, and *radial* half-edge operators respectively provided by Lage et al. [44].

We use the following function to create a wedge (object).

```
Wedge w(int a, int b) {return new Wedge(a,b);}
```

We use the following three ***basic wedge operators*** shown in Figure 11 (top right and bottom right):

```
Wedge m(Wedge w) {return w(w.b,w.a);}                    // mirror
Wedge n(Wedge w) {                                       // next
 int nc=m4(m4(w.b)+(even(w.a)?3:1));
 if(nc==m4(w.a)) {nc=m4(m4(w.b)+2);};
 return w(w.a,fc(w.a)+nc);}
Wedge o(Wedge w) { int na; int oc=O(w.b);                // opposite
 if(oc==c) {return null;};
  if(V(N(oc))==V(w.a)) {na=N(oc);}
  else if(V(P(oc))==V(w.a)) {na=P(oc);}
  else {na=N(N(oc));};
  return w(na,oc);};
```

The ***mirror*** wedge operator m() simply reverses the starting and ending corner indices.

The ***next*** wedge operator n(), based on whether corner w.a is odd or even, returns the proper next wedge. If we represent a tetrahedron by the corners 0, 1, 2 and 3, then by construction and our adopted tetrahedron orientation, (1,2,3) looks counter-clockwise from 0, (0,2,3) looks clockwise from 1, (0,1,3) looks counter-clockwise from 2 and (0,1,2) looks clockwise from 3. Hence, we can see that even corners c_e view face $f(c_e)$ as counter-clockwise and odd corners c_o view face $f(c_o)$ as clockwise. The n() operator returns the next counter-clockwise wedge, i.e. corner n(w).b is counter-clockwise relative to corner w.b when viewed from corner w.a. The ***opposite*** wedge operator o() uses the corner operator O(w.b) to identify the corner b opposite to w.b in an adjacent tetrahedron. However, there are **three wedges having b as the starting vertex**. We return the edge whose end-corner is at a vertex that is not bounding the tetrahedron of w. We could accelerate o() by caching the ***rotation number*** (Section 3.6.2) indicating which of the three wedges has b as starting vertex. Although we do not cache the rotation number for the VOT and SVOT, we will cache that rotation number for the SOT, as discussed in Section 3.6.

Also note that o() returns null when the wedge has no opposite. This software engineering decision facilitates the implementation of derived wedge operators by deferring the testing of border conditions. Our implementation of the m(), n() and o() operators (not shown here) returns null if a null wedge is received as input.

From these three basic wedge operators, we construct seven convenient ***derived wedge operators*** listed below and shown in Figure 11. For practice, we encourage the reader to visually verify their implementation using Figure 11.

```
Wedge p(Wedge w) {return n(n(w));}          // previous wedge
Wedge l(Wedge w) {return o(n(w));}              // left wedge
Wedge r(Wedge w) {return o(p(w));}             // right wedge
Wedge k(Wedge w) {return n(m(p(w)));}          // cross wedge
Wedge f(Wedge w) {return o(m(w));}           // forward wedge
Wedge sl(Wedge w) {return n(l(w));}        // swing left wedge
Wedge sr(Wedge w) {return p(r(w));}       // swing right wedge
```

3.4.1 Using wedge operators

To demonstrate the use of the wedge operator, we discuss here the VOT implementation of Brisson's swaps and of three common algorithms.

3.4.1.1 Brisson's swaps using wedge operators

Although we do not use Brissons's swap operators in our algorithms, we provide their implementation below in order to demonstrate the power and ease of use of our wedge operators. The 4-tuple (V,E,F,T) which was used above as context for Brisson's swaps defines an ordered list of 3 corners (a,b,c) of T such that V(a)=V, V(b) is the other vertex of E (other then V(a)), and V(c) is the third vertex of F. $swap_0$ and $swap_1$ are fairly straightforward. $swap_2$ returns the face sharing edge (V(a),V(b)) by utilizing the next and previous wedge operators to identify the corner in the tetrahedron which is not a, b or c. $swap_3$ returns the adjacent tetrahedron by utilizing the next, swing left and swing right wedge operators to identify the face connected tetrahedron.

```
class Triplet{int a, b, c;}
Triplet t3(int a, int b, int c)
 {return new Triplet(a,b,c);}
Triplet swap0(int a,int b,int c){return t3(b,a,c);}
Triplet swap1(int a,int b,int c){return t3(a,c,b);}
Triplet swap2(int a,int b,int c){ Wedge w=w(a,b);
 if(n(w).b==c){return t3(a, b, p(w).b);}
  else return t3(a, b, n(w).b);}
Triplet swap3(int a,int b,int c){ Wedge w=w(a,b);
 if(n(w).b==c){return t3(sr(w).a,sr(w).b,p(sr(w)).b);}
  else return t3(sl(w).a, sl(w).b, n(sl(w)).b);}
```

3.4.1.2 Swinging around an edge

Here, we explain how to visit all tetrahedra incident on an edge. Given a wedge w, we iteratively use the *swing left* operator until we return to w or reach a null wedge (meaning, we are outside of the mesh). If we reach the w, we are done. If we reach a null wedge, we repeat the process, but swinging right. The wedges we visit identify the tetrahedra incident on the given wedge and provide a starting reference for processing them.

```
Wedge swing(Wedge w) {                  //swing around wedge
   Wedge sw = w(w.a, w.b);                    //start wedge
   process(w);                              //process wedge
   while(w!=null){                    //not a boundary wedge
      if(eqW(sw, w)) break;           //reached start wedge
      w = sl(w);                                //swing left
      process(w);}                             //process wedge
```

```
 if(w==null) {                          //restart from starting wedge
    w = w(sw.a, sw.b);
    while(w!=null){ w = sr(w);                    //swing right
     process(w);}}}                              //process wedge

boolean eqW(Wedge u, Wedge w) {                   //equal wedges
      return (u.a==w.a && u.b==w.b);}             //u and w equal?
```

3.4.1.3 Visiting components of the boundary

Here, we explain how to traverse connected components of the boundary of a tetrahedral mesh using the VOT without building any auxiliary triangle mesh or other representation of that boundary. Such a traversal was exploited in tetstreamer [27] to traverse the triangulated surface separating the encoded tetrahedral from the others.

We can traverse a *shell* (connected manifold component) of a triangle mesh by starting from a corner c and recursively walking to neighboring triangles using the l(c) and r(c) *Corner Table* operators. We can either mark the visited triangles and use recursive calls, or mark visited vertices and triangles to avoid most recursive calls, as done in the Edgebreaker traversal [41]. The recursive version is listed below.

```
void visitshell(int c){                                 //visit shell
 if(!visited(t(c))){                                  //visited triangle?
   setVisited(t(c));                               //set triangle as visited
   visitshell(l(c));                                  //visit left triangle
   visitshell(r(c));}}                               //visit right triangle
```

Note that to each corner b of a border triangle of a tetrahedron mesh corresponds a unique wedge w=wedge(a,b). Using the analogy between wedge and corner operators, we can execute the above traversal algorithm and visit the faces of the connected component of the boundary of the mesh. All we need is the wedge counterparts lc(w) of the l(c) corner operator, where lc(w) implies the left corner operator on the boundary of the tetrhedral mesh. Likewise, rc(w) and oc(w) for r(c) and o(c) corner operators respectively. We provide the implementation of the lc(w), rc(w) and oc(w) operators below.

```
Wedge rc(Wedge w){return swing(p(m(w)));}     //right boundary
Wedge lc(Wedge w){return swing(n(k(w)));}      //left boundary
Wedge oc(Wedge w){return swing(m(k(w)));}          //opposite
Wedge swing(Wedge w){                  //swing around w until we..
 while(true){ if(B(k(w).a) break; w = sr(w);}       //.. hit boundary
 return k(w);}                                  //return proper wedge
```

3.4.1.4 Visiting the TST

Here we describe how to visit a Tetrahedron Spanning Tree for tetrahedral meshes, which is used in several compression techniques [16, 82].

We use the *right*, *left*, *opposite* and *forward* wedge operators. A temporary array of bits or flags is maintained to record the visited status of all tetrahedra. (We use the most significant bit of the Vertex Table to store them). The recursive version of the code is provided below.

```
void dfs(Wedge w) {                              //depth first traversal
 if(w!=null && !VisitedT(T(w.a))) {               //if tet not visited
  setVisitedT(T(w.a), true);                     //set visited status
  dfs(r(w)); dfs(l(w));                       //visit right and left tets
  dfs(o(w)); dfs(f(w));}              //visit opposite and forward tets
```

3.5 Sorted Vertex Opposite Table (SVOT)

3.5.1 Motivation

Various operations such as computing the average of the scalar values of the neighboring vertices off a given vertex requires accessing the star of the given vertex. As mentioned in Section 2.4.1 for triangle meshes, as suggested by Lage et al [44], one could store a vertex-to-corner lookup table which requires additional storage. As we did for triangle meshes, we sort the VOT to provide a direct access from each vertex to one of its corners without additional storage.

3.5.2 Proposed SVOT solution

The idea of the *Sorted Vertex Opposite Table* presented in Section 2.4.2 for triangle meshes extends trivially to tetrahedral meshes. For tetrahedral meshes, corner c(v) incident upon vertex v is 4*v. Narrow meshes are addressed in Section 3.5.5.

3.5.3 SVOT construction algorithm

The construction of the *SVOT* follows the one presented in Section 2.4.3. We map each vertex to a unique tetrahedron and sort the V Table based on the mapping.

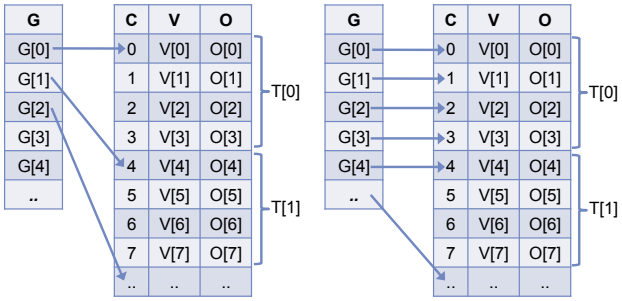

Figure 12: *The SVOT for tetrahedral meshes.*

3.5.3.1 Vertex to Tetrahedron Mapping

We extend the mapping algorithm listed in Section 2.4.3.1 to tetrahedral meshes. In the initialization step, with S as root tetrahedron, we choose T(l(w).b) as the first child tetrahedron, where w is the wedge from the first corner c to corner N(c) of S. We map V(c) to T(c) (root tetrahedron) and V(l(w).b) to T(l(w).b) (first child tetrahedron). We mark all vertices of S as visited.

In the termination step, we match the remaining three tetrahedra incident upon seed S (tetrahedra other than the first child tetrahedron) with 3 vertices of S (vertices other than V(c)), as in shown in Figure 14. These 3 tetrahedra will not be mapped to any vertex prior to the termination step. The reason being that we marked all vertices of S as being visited in the initialization step, and the 3 incident tetrahedra have been reached while coming from tetrahedra other than S hence will not be associated with a vertex (since their tip vertex was mapped to S during initialization and is no longer available to be associated with them).

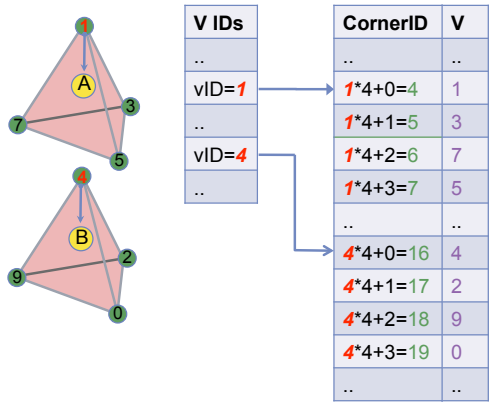

Figure 13: *Vertices 1 and 4 are mapped to tetrahedra A and B. Tetrahedron A consists of vertices (1,3,7,5) and tetrahedron B of (4,2,9,0). After sorting in SVOT, tetrahedron A is defined by corners (4,5,6,7) and tetrahedron B by corners (16,17,18,19) as vertex 1 maps to tetrahedron at 1st location and vertex 4 maps to tetrahedron at 4th location.*

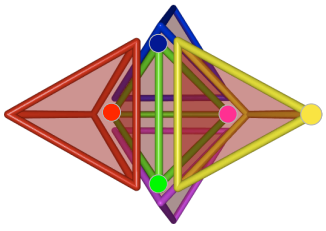

Figure 14: *The seed tetrahedron S (shown in green) with the corner c (shown as a green circle), its first neighbor T(l(w)) (shown in yellow), and the other three adjacent neighbors of S (shown in red, blue and magenta). Each tetrahedron shown is matched with a vertex of the same color. The other tetrahedra in the mesh are not shown.*

3.5.3.2 Sorting the V Table

We extend the sorting presented in Section 2.4.3.2 to tetrahedral meshes. We utilize the vertex-to-corner mapping to perform the sorting.

3.5.4 Traversing the star

Given an integer reference v to a vertex, the *SVOT* gives us direct access to the corresponding corner c(v), using:

```
int c(int v) {return v*4;}                    //for tetrahedral meshes
```

The corner and wedge operators for the *VOT* work without modification on the *SVOT*.

The i^{th} vertex is mapped to the $4*i^{th}$ corner in the SVOT. Notice, in Figure 13, in the SVOT, vertex 1 is located in the corner location 4*1=4, and vertex 4 is located in the corner location 4*4=16. To visit the star of the i^{th} vertex, we simply call star(4*i). Additionally, using the 4*i relation, we can traverse all tetrahedra incident on a vertex by performing a depth first traversal utilizing the *right*, *left* and *opposite* wedge operators. It is similar to depth first traversal except we do not use the forward wedge operator. If we are given the corner c then we recursively visit all the right, left and opposite wedges of w where w.a=c and w.b = N(c).

```
void star(int c) { Wedge w = w(c, N(c)); star(w);} //star of corner c
void star(Wedge w) {                                  //star traversal
 if(w!=null && !VisitedT(T(w.a))) {                  //if tet not visited
  setVisitedT(T(w.a), true);                         //set visited status
  process(T(w.a));                                   //process the tetrahedron
  star(r(w)); star(l(w)); star(o(w));}}              //visit neighbors
```

3.5.5 Special cases of narrow meshes

We follow a similar idea as presented in Section 2.4.4. The seed tetrahedron S has four vertex references and the vertex-to-corner mapping is as follows: for i<4, the i^{th} vertex maps to the i^{th} corner. For i>=4, the i^{th} vertex maps to the $((i-3)*4)^{th}$ corner. For multiple components with *m* multiple narrow meshes, we can place each of the *m* seed tetrahedron for each component as the first *m* tetrahedra in the V Table. This mapping has been shown in Figure 12, right.

3.6 Sorted Opposite Table (SOT)

The *VOT* and our *SVOT* variation each store 8 references per tetrahedron (4 to vertices and 4 to opposite corners). We discuss here an approach to reduce this storage to 4 references and 9 ***service bits*** per tetrahedron.

3.6.1 Construction of SOT:

We first construct the *SVOT*. We then obtain the *SOT* by eliminating the V table in the *SVOT* and by adding service bits. In the O Table, we store 9 *service bits* per tetrahedron; two *service bits* per corner (hence 8 bits per tetrahedron) to encode the *rotation number* (as described in Section 3.6.2) and 1 bit per tetrahedron for its visited state.

We adapt the approach of Section 2.5 to tetrahedral meshes. To visit the *star*, our operators use the O-table and the service bits. To traverse the *star*, we use the star traversal explained in section 3.5.4. During the traversal, we use the one *service bit* per tetrahedron to remember which tetrahedra were visited. We perform a second pass to erase them.

3.6.2 Rotation number:

The *rotation number* provides information about the relative orientation of two face-adjacent tetrahedra. Specifically, consider two tetrahedra t_1 with corner c and t_2 with corner O(c) that share a common face f(c). Imagine that we do not have the vertex references stored for the individual corners in f(c). There are three ways (see Figure 15) in which we can "glue" t_1 and t_2 at f. We can rotate t_1 clockwise, counter-clockwise, or not at all. Correspondingly, the *rotation number* rn(c) for corner c is either be 0, 1 or 2. The rotation number is computed by using the V table in SVOT. We cache the rotation number, rn(c) for each corner c as the most significant bits of each entry in the opposites table. When using 32 bit references, we can still process meshes with upto ~250 million tetrahedra. The code below computes the rotation number for a given corner.

```
int RotationNumber(int c) {          //rotation number for corner c
 Wedge w = w(O(c), N(O(c)));                    //create a wedge
 if(V(N(c))==V(w.b)) return 0;                 //no rotation required
 if(V(N(c))==V((n(w)).b)) return 1;             //1 rotation required
 return 2;}                                    // 2 rotations required
```

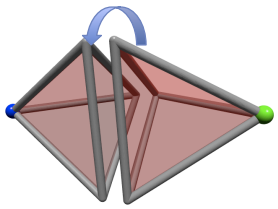

Figure 15: *If the vertex references of the shared face is not known, then there are three ways in which the two tetrahedra could be "twisted" and "glued". We refer to the correct "twisting" as the rotation number.*

3.6.3 Determining vertex references:

We explain here how the vertex for an arbitrary corner b can be inferred from the O table of the SOT and from the *service bits*.

The idea is to traverse the ***star*** (see section 3.5.4) of V(b) to determine the corners b_i incident on V(b). We must do that of course without knowing V(b), since V(b) is the desired result. Traversing the star is possible by using the *left*, *right* and *opposite* wedge operators as these wedge operators require only the O Table and service bits of the SOT. The traversal stops when we find a ***matching*** tetrahedron $T<n_V$.

Finding the vertex reference can require that we visit at most d tetrahedra, where d is the valence of the vertex v. Our experiments indicate that we need to visit, on average, about 13.5 tetrahedra (the valence is, on average, 26).

3.6.4 Special cases of narrow components

The special cases of narrow meshes generalizes from the *SVOT* (Section 3.5.5) to the *SOT*. For narrow meshes, the vertex references of a corner is found by traversing the star of a corner until a vertex-to-corner mapping is found. The mapping provides the vertex reference as defined in Section 3.5.5.

3.6.5 Wedge operators on SOT

In the SOT, we need to redefine the V() corner operator and the o() wedge operator. All other corner operators and wedge operators for tetrahedral meshes from the *VOT* remain the same.

For tetrahedral meshes, in the original *VOT* implementation, the *o*() wedge operator utilizes the V Table, but since the *SOT* doesn't store the V table, we rely on the *rotation number* to determine the proper o() wedge operator. The o() wedge operator takes constant steps for each call.

The V() operator traverses the star (as described in Section 3.5.4) to determine the vertex reference.

```
int V(int c) {                          //returns SOT vertex id
 Wedge w = w(c, N(c));                    //create wedge w
 return StarV(w);}                  //utilize star to get vertex id

int StarV(Wedge w) {               //star to determine vertex id
 int rv = -1;                             //default value
 if(w!=null) {                          //if wedge exists
  int t = T(w.a);                       //tetrahedron id
  if(!VisitedT(t)) {                 //if tetrahedron not visited
   setVisitedT(t, true);                //mark as visited
   if(t<v && m4(w.a)==0) {rv = t;}   //if first corner and t<|G|
   if(rv==-1) v = StarV(r(w));          //visit right wedge
   if(rv==-1) v = StarV(l(w));          //visit left wedge
   if(rv==-1) v = StarV(o(w));}}        //visit opposite wedge
 return rv;}                            //return vertex id


Wedge o(Wedge w) {                 //SOT opposite operator
 if(w==null) return null;             //wedge does not exist
 if(O(c)==c) return null;             //wedge does not exist

int c = w.b;                                    //corner c
 Wedge ow = w(O(c), N(oc));                  //no rotation
 if(rn(c)==1) ow = n(ow, tm);                 //1 rotation
 if(rn(c)==2) ow = p(ow, tm);                 //2 rotation

Wedge cw = w(c, N(c));                       //no rotation
 if(cw.b==w.a) {return w(ow.b, ow.a);}       //no aligning
```

```
 if(n(cw, tm).b==w.a) { return w(p(ow, tm).b, ow.a);}//align next
return w(n(ow, tm).b, ow.a);}                        //align with prev
```

3.7 Summary

The extension of the SVOT and SOT data structures from triangle meshes to tetrahedral meshes is trivial. However, the new v function for SOT is somehow more complicated and adds a time overhead because a call to v performs on average 27 operations that each access the next tetrahedron. Nevertheless, the SOT requires only 4 references per tetrahedron to encode the connectivity (adjacency, star and ordering) information.

4. CONCLUSION

The *VOT* representation of triangle (resp. tetrahedral) meshes requires 6 references per triangle (resp. 8 references per tetrahedron). We propose two variations: (1) The *SVOT* affords references from a vertex to one of its incident corners without increasing storage. It permits to access all incident and adjacent cells to a corner, a vertex, or a triangle (resp. tetrahedron) with a constant cost per cell. (2) The *SOT* further reduces storage cost to 3 references per triangle (resp. 4 references and 9 service bits per tetrahedron), but makes the computational cost of accessing neighboring cells proportional to the valence of a common vertex.

To facilitate the use of these new representations for tetrahedral meshes, we introduce a small set of powerful wedge operators for querying and traversing the tetrahedral mesh and provide efficient implementations that work directly off the *VOT, SVOT or SOT*. These wedge operators are a natural and intuitive extension to tetrahedral meshes of the familiar *Corner Table* operators originally developed for triangle meshes. We illustrate their power by providing reasonably simple source code for several common algorithms that process tetrahedral meshes.

5. REFERENCES

1. Alik, H. and T.J.R. Hughes, *Finite element method for piezoelectric vibration.* International Journal for Numerical Methods in Engineering, 1970. **2**(2): p. 151-157.
2. Caendish, J.C., D.A. Field, and W.H. Frey, *An apporach to automatic three-dimensional finite element mesh generation.* International Journal for Numerical Methods in Engineering, 1985. **21**(2): p. 329-347.
3. Garimella, R.V., *Mesh data structure selection for mesh generation and FEA applications*, in *International Journal for Numerical Methods in Engineering*. 2002. p. 451-478.
4. Sambridge, M., J. Braun, and H. McQueen, *Geophysical parametrization and interpolation of irregular data using natural neighbours.* Geophysical Journal International, 1995. **122**(3): p. 837-857.
5. Boissonnat, J.-D., *Shape reconstruction from planar cross sections.* Comput. Vision Graph. Image Process., 1988. **44**(1): p. 1-29.
6. Hartmann, U. and F. Kruggel, *A Fast Algorithm for Generating Large Tetrahedral 3D Finite Element Meshes from Magnetic Resonance Tomograms*, in *Proceedings of the IEEE Workshop on Biomedical Image Analysis*. 1998, IEEE Computer Society.
7. Pescatore, J., L. Garnero, and I. Bloch, *Tetrahedral finite element meshes of head tissues from MRI for the MEG/EEG forward problem*, in *12th Scandinavian Conference on Image Analysis*. 2001. p. 71-80.
8. Aurenhammer, F., *Voronoi diagrams- a survey of a fundamental geometric data structure.* ACM Comput. Surv., 1991. **23**(3): p. 345-405.
9. Bruzzone, E. and L.D. Floriani, *Two data structures for building tetrahedralizations.* Vis. Comput., 1990. **6**(5): p. 266-283.
10. Dobkin, D.P. and M.J. Laszlo, *Primitives for the manipulation of three-dimensional subdivisions*, in *Proceedings of the third annual symposium on Computational geometry*. 1987, ACM: Waterloo, Ontario, Canada.
11. Dobkin, D.P. and M.J. Laszlo, *Primitives for the manipulation of three-dimensional subdivisions*, in *Algorithmica*. 1989. p. 3-32.
12. Guibas, L. and J. Stolfi, *Primitives for the manipulation of general subdivisions and the computation of Voronoi.* ACM Trans. Graph., 1985. **4**(2): p. 74-123.
13. Lopes, H. and G. Tavares, *Structural operators for modeling 3-manifolds*, in *Proceedings of the fourth ACM symposium on Solid modeling and applications*. 1997, ACM: Atlanta, Georgia, United States.
14. Gregorski, B., et al., *Interactive view-dependent rendering of large isosurfaces*, in *Proceedings of the conference on Visualization '02*. 2002, IEEE Computer Society: Boston, Massachusetts.
15. Lindstrom, P., *Out-of-core simplification of large polygonal models*, in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, ACM Press/Addison-Wesley Publishing Co. p. 259-262.
16. Gumhold, S., S. Guthe, and W. Straser, *Tetrahedral mesh compression with the cut-border machine*, in *Proceedings of the conference on Visualization '99: celebrating ten years*. 1999, IEEE Computer Society Press: San Francisco, California, United States.
17. King, D., J. Rossignac, and A. Szymczak, *Connectivity Compression for Irregular Quadrilateral Meshes.* GVU Tech Report GIT-GVU-99-36. PDF., 1999.
18. Pajarola, R., J. Rossignac, and A. Szymczak, *Implant sprays: compression of progressive tetrahedral mesh connectivity*, in *Proceedings of the conference on Visualization '99: celebrating ten years*. 1999, IEEE Computer Society Press: San Francisco, California, United States.
19. Botsch, M., et al., *Geometric modeling based on triangle meshes*, in *Course Notes, ACM SIGGRAPH 2006*. 2006, ACM Press.

20. Isenburg, M., et al., *Streaming compression of tetrahedral volume meshes*, in *Proceedings of Graphics Interface 2006*. 2006, Canadian Information Processing Society: Quebec, Canada.
21. Ueng, S.-K. and K. Sikorski, *An out-of-core method for computing connectivities of large unstructured meshes*, in *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*. 2002, Eurographics Association: Blaubeuren, Germany.
22. Chopra, P. and J. Meyer, *TetFusion: an algorithm for rapid tetrahedral mesh simplification*, in *Proceedings of the conference on Visualization '02*. 2002, IEEE Computer Society: Boston, Massachusetts.
23. Vo, H.T., et al., *Streaming Simplification of Tetrahedral Meshes.* IEEE Transactions on Visualization and Computer Graphics, 2007. **13**(1): p. 145-155.
24. Liu, A. and B. Joe, *Quality local refinement of tetrahedral meshes based on 8-subtetrahedron subdivision.* Math. Comput., 1996. **65**(215): p. 1183-1200.
25. Shewchuk, J.R., *Tetrahedral mesh generation by Delaunay refinement*, in *Proceedings of the fourteenth annual symposium on Computational geometry*. 1998, ACM: Minneapolis, Minnesota, United States.
26. Liu, Y. and J. Snoeyink, *A comparison of five implementations of 3d Delaunay tessellation*, in *Combinatorial and Computational Geometry, MSRI series*. 2005. p. 439-458.
27. Bischoff, U. and J. Rossignac, *TetStreamer: Compressed Back-to-Front Transmission of Delaunay Tetrahedra Meshes*, in *Proceedings of the Data Compression Conference*. 2005, IEEE Computer Society.
28. Rossignac, J., P. Borrel, and L. Nackman. *Interactive Design with Sequences of Parameterized Transformations*. in *Eurographics Workshop on Intelligent CAD Systems: Implementation Issues*. 1988. Veldhoven, The Netherlands.
29. Rossignac, J., P. Borrel, and L. Nackman. *Procedural Models for Design and Fabrication*. in *Proc. of the MIT Sea Grant Symposium*. 1990. Boston: Hemisphere Publishing Co.
30. Snyder, J.M., *Generative Modeling for Computer Graphics and CAD: Symbolic Shape Design Using Interval Analysis*. 1992, San Diego: Academic Press.
31. Stingy, G., *Introduction to shape and shape grammars.* Environment and Planning B, 1980. **7**(3): p. 343 – 351.
32. Benjamin, S., P. Alexander, and S. Christophe, *Constructive modeling of FRep solids using spline volumes*, in *Proceedings of the sixth ACM symposium on Solid modeling and applications*. 2001, ACM: Ann Arbor, Michigan, United States. p. 321-322.
33. Cartwright, R., et al., *Web-Based Shape Modeling with Hyperfun.* IEEE Comput. Graph. Appl., 2005. **25**(2): p. 60-69.
34. Rossignac, J. and A. Requicha, *Solid Modeling*, in *Encyclopedia of Electrical and Electronics Engineering*, J.G. Webster, Editor. 1999, John Wiley & Sons. p. 658-672.
35. Rossignac, J. and A. Requicha, *Constructive Non-Regularized Geometry.* Computer-Aided Design, 1991. **23**(1): p. 21-32.
36. Hable, J. and J. Rossignac, *Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes.* ACM Transactions on Graphics, 2005. **24**(3): p. 1024-1031.
37. Hable, J. and J. Rossignac, *CST: Constructive Solid Trimming for rendering BReps and CSG models.* IEEE Trans. on Visualization and Computer Graphics (TVCG), 2007. **13**(5): p. 1004-1014.
38. Rossignac, J. and M. O'Connor. *SGC: A Dimension-Independent Model for Pointsets with Internal Structures and Incomplete Boundaries*. in *IFIP Workshop on CAD/CAM*. 1989. North-Holland.
39. Rossignac, J., *Through the cracks of the solid modeling milestone*, in *From object modelling to advanced visualization*, S. Coquillart, W. Strasser, and P. Stucki, Editors. 1994, Springer Verlag. p. 1-75.
40. Brisson, E., *Representing geometric structures in d dimensions: topology and order*, in *Proceedings of the fifth annual symposium on Computational geometry*. 1989, ACM: Saarbruchen, West Germany. p. 218-227.
41. Rossignac, J., A. Safonova, and A. Szymczak, *3D Compression Made Simple: Edgebreaker on a Corner-Table*, in *3D Compression Made Simple: Edgebreaker with Zip&Wrap on a Corner-Table*. 2001, IEEE Computer Society.
42. Rossignac, J., A. Safonova, and A. Szymczak, *Edgebreaker on a Corner Table: A simple technique for representing and compressing triangulated surfaces*, in *Hierarchical and Geometrical Methods in Scientific Visualization*. 2003. p. 41-50.
43. Rossignac, J., *3D Mesh Compression*, in *The Visualization Handbook*, C. Hansen and C. Johnson, Editors. 2006, Academic Press.
44. Lage, M., et al., *CHF: A Scalable Topological Data Structure for Tetrahedral Meshes*, in *Proceedings of the XVIII Brazilian Symposium on Computer Graphics and Image Processing*. 2005, IEEE Computer Society.
45. Gurung, T. and J. Rossignac, *SOT: Compact Representation for Tetrahedral Meshes*, in *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*. 2009, ACM: San Francisco, California. p. 79-88.
46. Baumgart, B.G., *Winged edge polyhedron representation*. 1972, Stanford University.
47. Baumgart, B.G., *A polyhedron representation for computer vision*, in *Proceedings of the May 19-22, 1975, national computer conference and exposition*. 1975, ACM: Anaheim, California.
48. Mantyla, M., *Introduction to Solid Modeling*. 1988: W. H. Freeman & Co. 144.
49. Lienhardt, P., *Subdivisions of n-dimensional spaces and n-dimensional generalized maps*, in *Proceedings of the fifth annual symposium on Computational geometry*. 1989, ACM: Saarbruchen, West Germany.
50. Weiler, K., *The radial-edge data structure: a topological representation for non-manifold geometric boundary modeling.* Geometric Modeling for CAD Appl., 1988: p. 3-36.
51. Floriani, L.D. and A. Hui, *Data structures for simplicial complexes: an analysis and a comparison*, in *Proceedings of the third Eurographics symposium on Geometry processing*. 2005, Eurographics Association: Vienna, Austria.
52. Joy, K.I., J. Legakis, and R. MacCracken, *Data Structures for Multiresolution Representation of Unstructured Meshes*, in *Hierarchical Approximation and Geometric Methods for Scientific Visualization*. 2002.

53. Kallmann, M. and D. Thalmann, *Star-vertices: a compact representation for planar meshes with adjacency information.* Journal of Graphics Tools, 2001. **6**(1): p. 7-18.
54. Campagna, S., L. Kobbelt, and H.-P. Seidel, *Directed edges-A scalable representation for triangle meshes.* Journal of Graphics Tools, 1998. **3**(4): p. 1-11.
55. Sander, P.V., D. Nehab, and J. Barczak, *Fast triangle reordering for vertex locality and reduced overdraw.* ACM Trans. Graph., 2007. **26**(3): p. 89.
56. Sander, P.V., et al., *Efficient traversal of mesh edges using adjacency primitives.* ACM Trans. Graph., 2008. **27**(5): p. 1-9.
57. Rossignac, J., *Surface simplification and 3D geometry compression*, in *The Handbook of Discrete and Computational Geometry (2nd edition)*, Goodman and O'Rourke, Editors. 2004, CRC Press.
58. Cormen, T.H., et al., *Introduction to Algorithms (2nd ed.)*. 2001: MIT Press and McGraw-Hill.
59. Ueng, S.-K. and K. Sikorski, *A Note on a Linear Time Algorithm for Constructing Adjacency Graphs of 3D FEA Data*, in *The Visual Computer*. 1996. p. 445-450.
60. Tautges, T.J., et al., *The Sandia Mesh Database Component (MDB)*, in *Proceedings of the Seventh Us National Congress on Computational Mechanics*. 2003.
61. Garimella, R.V., *MSTK - A Flexible Infrastructure Library for Developing Mesh Based Applications*, in *Proceedings of 13th International Meshing Roundtable*. 2004. p. 203-212.
62. Remacle, J.F., B.K. Karamete, and M.S. Shephard, *Algorithm Oriented Mesh Database.* Proceedings of the 9th International Meshing Roundtable, 2000: p. 349-359.
63. Beall, M.W. and M.S. Shephard, *A General Topology-based Mesh Data Structure.* International Journal for Numerical Methods in Engineering, 1997. **40**(9): p. 1573-1596.
64. Shephard, M.S. and P.M. Finnigan, *Integration of geometric modeling and advanced finite element preprocessing*, in *Finite Elements in Analysis and Design*. 1988. p. 147-162.
65. Edelsbrunner, H., *Geometry and Topology for Mesh Generation*. 2001: Cambridge University Press.
66. Garimella, R.V. and M.S. Shephard, *Tetrahedral Mesh Generation With Multiple Elements Through the Thickness*, in *International Meshing Roundtable*. 1995. p. 321-333.
67. Joe, B., *GEOMPACK - A Software Package for the Generation of Meshes Using Geometric Algorithms*, in *Advances in Engineering Software*. 1991. p. 325-331.
68. TetMesh, *TetMesh - GHS3D, Ver. 3.1.* Tech. report INRIA/SIMULOG, 2001.
69. LaGrit, *LaGriT - Los Alamos Grid Toolbox.* Tech. report Los Alamos National Laboratory, 1995.
70. CUBIT, *CUBIT Mesh Generation Toolkit.* Tech. report Sandia National Laboratories, 2001.
71. Chen, D., et al., *Geometry compression of tetrahedral meshes using optimized prediction.* European Conference on Signal Processing, 2005.
72. Yang, C.-K., T. Mitra, and T.-C. Chiueh, *On-the-Fly rendering of losslessly compressed irregular volume data*, in *Proceedings of the conference on Visualization '00*. 2000, IEEE Computer Society Press: Salt Lake City, Utah, United States.
73. Staadt, O.G. and M.H. Gross, *Progressive tetrahedralizations*, in *Proceedings of the conference on Visualization '98*. 1998, IEEE Computer Society Press: Research Triangle Park, North Carolina, United States.
74. Yoon, S.-e. and P. Lindstrom, *Random-Accessible Compressed Triangle Meshes.* IEEE Transactions on Visualization and Computer Graphics, 2007. **13**(6): p. 1536-1543.
75. Trotts, I.J., B. Hamann, and K.I. Joy, *Simplification of Tetrahedral Meshes with Error Bounds.* IEEE Transactions on Visualization and Computer Graphics, 1999. **5**(3): p. 224-237.
76. Cignoni, P., et al., *Simplification of Tetrahedral meshes with accurate error evaluation*, in *Proceedings of the conference on Visualization '00*. 2000, IEEE Computer Society Press: Salt Lake City, Utah, United States.
77. Danovaro, E., et al., *Multiresolution Tetrahedral Meshes: An Analysis and a Comparison*, in *Proceedings of the Shape Modeling International 2002 (SMI'02)*. 2002, IEEE Computer Society.
78. Cignoni, P., et al., *Selective Refinement Queries for Volume Visualization of Unstructured Tetrahedral Meshes.* IEEE Transactions on Visualization and Computer Graphics, 2004. **10**(1): p. 29-45.
79. Cutler, B., J. Dorsey, and L. McMillan, *Simplification and improvement of tetrahedral models for simulation*, in *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. 2004, ACM: Nice, France.
80. Danovaro, E., et al., *The Half-Edge Tree: A Compact Data Structure for Level-of-Detail Tetrahedral Meshes*, in *Proceedings of the International Conference on Shape Modeling and Applications 2005*. 2005, IEEE Computer Society.
81. Sondershaus, R. and W. Straser, *View-dependent tetrahedral meshing and rendering*, in *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. 2005, ACM: Dunedin, New Zealand.
82. Szymczak, A. and J. Rossignac, *Grow & fold: compression of tetrahedral meshes*, in *Proceedings of the fifth ACM symposium on Solid modeling and applications*. 1999, ACM: Ann Arbor, Michigan, United States.
83. Weiler, M., et al., *Texture-Encoded Tetrahedral Strips*, in *Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics*. 2004, IEEE Computer Society.